

Testing the Real-time features of POSIX

Prepared by

Andrew Josey

The Open Group

- A White Paper-

Revision 1
June 2000

The distribution and review of this document is limited to The Open Group and Real-time and Embedded Systems Forum

The Open Group
Apex Plaza,
Forbury Road,
Reading RG1 1AX
England

Tel: +44 118 950 8311 x2250
Fax: +44 118 950 0110

Project Manager: Andrew Josey Email: a.josey@opengroup.org
Business Contact: Robert Noyes Email: r.noyes@opengroup.org

Contents

Objective

This document is a white paper on testing for Realtime POSIX^{®1}. It provides a high level description of the current test suites that are available and describes the test methodology, the functional coverage and the test architecture.

Scope

The main body of this document is split into 2 parts, as outlined below.

Part 1: Test Development Process and Infrastructure

An overview of the test development process and the common test framework to be utilized for the Realtime test suites, which is provided by the Test Environment Toolkit (TET) and the VSXgen framework. Details are provided of:

- Methodology
- Assertions
- Coverage
- Acceptance Criteria
- Infrastructure
- Report facilities

Part 2: A proposal to test Embedded POSIX Realtime systems

Details of a technical proposal to test Embedded POSIX Real-time systems. This outlines the modifications needed for the existing test framework and tests to add coverage for the POSIX 1003.13 profiles. These include:

- An overview of POSIX 1003.13 Units of functionality
- TET modifications
- VSXgen modifications
- Other issues

¹ POSIX is a registered trademark of the IEEE.

Definitions and Abbreviations

Definitions

The following terminology is used throughout this document to describe pieces of the Realtime POSIX amendments:

.1d,	POSIX 1003.1d-1999,	Additional Realtime Extensions
.1h,	POSIX P1003.1h,	Services for Reliable, Available and Serviceable Systems
.1j,	POSIX 1003.1j-2000,	Advanced Realtime Extensions
.1m,	POSIX P1003.1m,	Checkpoint/Restart Interface
.1q,	POSIX P1003.1q,	Tracing
.13,	POSIX 1003.13-1998,	Realtime Application Support Profile

Abbreviations

The following abbreviations are used:

TET	The Test Environment Toolkit – a multi-platform test scaffold used by The Open Group to develop both non-distributed and distributed test suites.
VSXgen	The test framework built on the <i>Test Environment Toolkit</i> to provide a common infrastructure for modular test suites for POSIX and other UNIX-based standards.

References

Normative

IEEE Std 1003.1d-1999, IEEE Standard for Information Technology - Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) - Amendment x: Additional Realtime Extensions [C Language].

IEEE Std 1003.1j-2000, IEEE Standard for Information Technology - Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) - Amendment x: Advanced Realtime Extensions [C Language].

IEEE Std 1003.13-1998, IEEE Standard for Information Technology--Standardized Application Environment Profile (AEP) -- POSIX® Realtime Application Support. ISBN 0-7381-0178-8.

IEEE Std 1003.3-1991, IEEE Standard for Information Technology - Test Methods for Measuring Conformance to POSIX. ISBN 1-55937-104-8.

Informative

Draft IEEE Std P1003.1h-199x, IEEE Standard for Information Technology - Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) - System API Extensions - Services for Reliable, Available, and Serviceable Systems [C Language] - Amendment. Draft 5, July 1999.

Draft IEEE Std P1003.1m-199x, IEEE Standard for Information Technology - Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) - Amendment m: Checkpoint/Restart Interface [C Language]. Draft 2, November 1998.

Draft IEEE Std P1003.1q-199x, IEEE Standard for Information Technology - Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) - Amendment q: Tracing [C Language]. Draft 5, July 1999.

The Test Environment Toolkit, Architectural, Functional, and Interface Specification, Revision 4.5, X/Open Company Ltd, The Open Software Foundation and UNIX International, 1992.
<http://tetworks.opengroup.org/pdfdocs/tetspec.pdf>

TETware User Guide, Revision 1.2, TET3-UG-1.2. The Open Group, 1998.
<http://tetworks.opengroup.org/documents/3.3/uguide.pdf>

TETware Programmers Guide, Revision 1.2, TET3-PG-1.2. The Open Group, 1998.
<http://tetworks.opengroup.org/documents/3.3/pguide.pdf>

The Open Group, Alpha and Beta Test Activity Procedures.
<http://www.opengroup.org/testing/testprocs/>

Part 1

Test Development Process and Infrastructure

1 Test Development Process and Infrastructure

The processes and infrastructure used by The Open Group in its test developments are described in this section.

1.1 Test Development Methodology

The Open Group has developed a set of procedures for ensuring timely, quality, correctness, fitness-for-purpose test suite development, based on experience gained through over twelve years of X/Open test suite development. This involves a multi-step process where the objectivity and traceability of test code to a written specification are the key. It is the written specification (in this case the relevant POSIX standard) that is the definitive reference point. In cases of doubt assertions and tests are secondary and defer to the specification. Quality, correctness and fitness-for-purpose are achieved through a methodology that places regular checkpoints and peer-review feedback within the process.

1.1.1 Process overview

The process for developing the tests for POSIX is as follows:

- A test specification is developed
- A set of formal test assertions is developed.
- Test suites based on the specification and the assertions are developed and taken through a controlled development cycle before general availability.
- After completion of the test suites, the test suites are moved to support and maintenance mode where customer support questions, and interpretations are processed, and regular test suite updates made.

The Open Group has developed a set of criteria that the test specification, assertions and the test suite need to be meet in order to be accepted. These are known as *The Open Group Alpha and Beta Test Procedures*. At regular points formal reviews are held and feedback is obtained from the customers and specification owners to verify correct assumptions. A technical expert group is used to advise on specification interpretations during development (questions from developers) during Beta (issues raised by beta testers) and during test suite maintenance. Such issues are dealt with in a way that ensures an archived set of referencable interpretations for other test suite users and potential brand holders

1.1.2 Test Assertions

The creation of a set of test assertions is necessary to bridge the gap between natural language specification and test suite code. The goal of the test suite code is to produce an executable representation of the requirements of the specification under test.

The Open Group adopts an assertion-based technique for test specifications. This follows the IEEE Standard for Test Methods (1003.3). An assertion is developed for each definitive statement in the specification under test. Each assertion is designed to determine whether the statement is true or false for the implementation under test.

1.1.3 Testing Levels

POSIX.3 defines three levels of testing:

1. Exhaustive - Full testing of all aspects of a specification requirement.
2. Thorough - Testing sufficient to validate that the implementation under test matches a specification requirement without exhaustive testing.
3. Identification - Testing that the implementation-under-test provides the mechanism necessary to meet the specification requirement.

Exhaustive testing are used for the POSIX Realtime tests where practical. However in many areas it is not practical to do so, due to the time and system resources required. In such cases the test suites will perform thorough testing which serves as a fair compromise in the coverage/cost ratio. Identification testing will only be used if other methods prove to be impractical, or if the specification requires it.

1.1.4 Reviews

Once the test suite specification has been developed it will be validated to ensure that it is a correct representation of the requirements of the specification under test. The Open Group has developed methods for managing review feedback that allow development to proceed in an objective and timely fashion. The same method will be applied to the test assertions.

1.1.5 Test Suite Acceptance

The procedure for test suite acceptance will involve two review cycles:

A test suite snapshot will be delivered consisting of a complete test framework with over 50% of tests complete. A review at this early stage will allow validation of the general test approach, and issues to be identified with the assertions and underlying specification.

A Beta test cycle will provide the formal review of correctness once the test suite has been fully developed. The Open Group has developed procedures for formal acceptance of test suites where severity and numbers of problem reports are logged and used to objectively determine acceptance.

1.1.6 Test Suite Maintenance

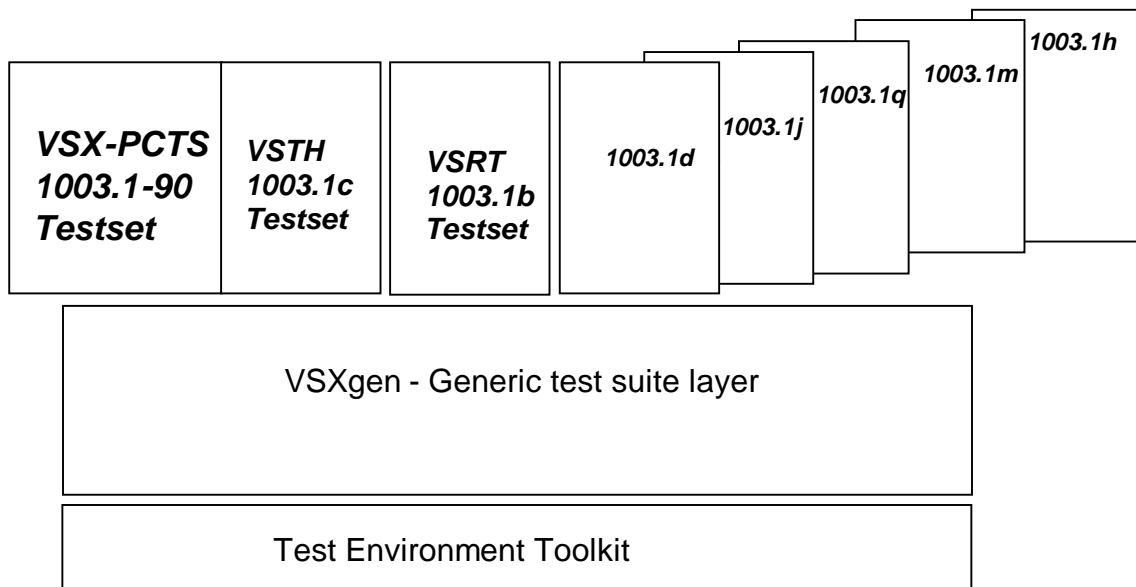
Once the test development is complete, subject to funding the test suites will go into a maintenance cycle. A support service will be established using the World Wide Web and electronic mail. A database of sanitised interpretations will be maintained that can be searched by other test suite users. Regular maintenance releases will be produced up to twice a year if necessary, addressing customer issues identified by using the test suite live.

1.2 Test Suite Infrastructure

1.2.1 Test Suite Architecture

The Open Group has a portfolio of test suites that cover the whole of POSIX 1003.1-1996, including POSIX 1003.1b (realtime) and POSIX 1003.1c (threads). Presently in development are tests for POSIX 1003.1d, POSIX 1003.1j and POSIX 1003.1q. All these test suites are run using the Test Environment Toolkit test harness and the VSXgen test framework.

The following diagram represents the structure of the existing test suites. A description of the modules in the architecture is provided in the following sections



The Open Group VSXgen-based Test Suites

1.2.2 The Test Environment Toolkit (TET)

1.2.2.1 TET Overview

The Test Environment Toolkit (TET) is a multi-platform uniform test scaffold, into which non-distributed and distributed test suites can be incorporated. It allows production of test suites sharing a common interface, therefore promoting sharing of test suites within organizations as well as between different organizations. Standardization of the test methodology and tools allows testing efforts to focus away from the harness and tools, thus increasing efficiency and productivity.

TET is being used in a wide diversity of automated testing applications, ranging from standards API conformance testing, network computer testing, performance and stress testing, verification of secure electronic transactions, to distributed cross platform applications.

1.2.2.2 TET Versions

TET may be built in two versions: One called TET-Lite for supporting non-distributed testing on a single computer system. The other called Distributed TET, which is able to process both distributed and non-distributed test cases on the local system or one or more remote systems. The POSIX Realtime test suites on platforms fully supporting POSIX 1003.1-1990 will use TET-Lite. The test suites to be developed for POSIX .13, as proposed in the next section of this document, will use a modified version of distributed TET.

1.2.2.3 Features and facilities

TET provides facilities to execute test cases in several ways as follows:

- Execution of sequences of test cases in a repeatable manner
- Execution of a single test case selected at random from a list of test cases
- Execution of test cases in parallel
- Sequences of the above elements executing a specified number of times or until some time period has expired

1.2.3 The VSXgen Framework

1.2.3.1 VSXgen Overview

VSXgen is an operating system test framework that allows tests for standard operating system APIs to be abstracted away from the framework itself. VSXgen builds upon the Test Environment Toolkit. This permits a common look and feel to test suites that are developed to reuse the framework, reduced development time, the ability to run the tests standalone as individual test suites, as well as to integrate them into a meta test suite.

For users familiar with installing and running one VSXgen based test suite, there is a minimal learning curve to running another test suite based on VSXgen. The framework itself is based on the core of the X/Open test suite VSX4, and is proven with over 10 years of use in the industry.

1.2.3.2 Current test subsets using VSXgen

The test suites supported as test subsets under the VSXgen structure are (as of June 2000):

- VSRT - Realtime (1003.1b/.1i)
- VSTH - Threads (1003.1c/Aspen threads)
- VSX-PCTS – The core tests for POSIX.1-1990
- VSX4 - The VSX4 test suite (POSIX.1/XPG4 extensions)
- VSX5 - The Large File, Dynamic Linking and ISO MSE tests
- VSU5 – The UNIX system extensions test suite
- VSRTE – Additional Realtime Extension (1003.1d)
- VSART – Advanced Realtime Extension (1003.1j)
- VSTRC – Tracing (1003.1q)

1.2.4 Report Writer Facilities

A formal report writer is provided with VSXgen that outputs a report suitable for formal certification activities. The report writer also includes facilities to aid test developers such as summarizing results for a test section or test, or comparative reporting of test journals.

This page is intentionally blank

Part 2

Proposal to Test Embedded POSIX Realtime Systems

2 Proposal to Test Embedded POSIX Realtime Systems

Contents

2.1	Introduction.....	12
2.2	Proposed New Work Packages	12
2.2.1	Work Package Breakdown.....	13
2.3	Technical Overview for Proposed Embedded Realtime System Testing	13
2.3.1	Overview of POSIX 1003.13 Profiles	13
2.3.2	Overview of POSIX 1003.13 Units of Functionality	14
2.4	TET Modifications.....	14
2.4.1	Host and target systems.....	14
2.4.2	TET version.....	14
2.4.3	Host system requirements	15
2.4.4	Test case launcher	15
2.4.5	Target system software.....	15
2.4.6	Communication between host and target systems	15
2.4.7	API versions	16
2.4.8	Names of the TCM and API components	16
2.4.9	User-defined functions	17
2.4.10	TCC - TCM interface	17
2.4.11	Making journal entries	18
2.4.12	Accessing configuration variables.....	18
2.4.13	Detecting test case malfunction	19
2.4.14	Test case termination.....	19
2.5	VSXgen Modifications	19
2.5.1	Changes Needed for Cross-compilation of VSXgen.....	19
2.5.2	Changes Needed to VSXgen to Support Testing of POSIX.13.....	22
2.5.3	Issues with POSIX .13.....	24
2.6	Test Suite Modifications	25
2.6.1	Categorize All POSIX Assertions	25
2.6.2	Determine Testability and Strategies of Applicable Assertions.....	25
2.6.3	Implement New Strategies for Testable Assertions Where Required.....	26
2.7	Current Commercial Practice	26
2.7.1	Product Diversity	26
2.7.2	Development environments	26
2.7.3	Process Execution.....	26

2.1 Introduction

The Open Group proposes to implement test suites for the C language bindings across all profiles defined in POSIX 1003.13. The strategy that The Open Group proposes for test suite development is to modify the existing Open Group test suites for POSIX .1, .1b, .1c and .2 so that the portions appropriate for a given profile can be run in an environment conforming to that profile. Thus the proposed test suites are based on an existing test framework and test suites that have been used and proved in the industry over the last ten years.

The first part of this section proposes the work packages for the test suite development. The second part outlines the technical details for the .13 packages.

2.2 Proposed New Work Packages

The work packages proposed in this section cover the remainder of the work packages specified in the original Open Group proposal that are not covered by the current contract (test suite development for 1003.1h and .1m), and new work packages to provide test suites for the POSIX 1003.13 Application Environment Profiles 51 to 54.

2.2.1 Work Package Breakdown

The propose work packages are listed below, numbered to correspond with the proposed work packages in the previous proposal. The technical rationale behind the .13 packages is given later in the section.

Work Package	Work Package Description
Work Packages cover test suites for the POSIX .13 standard	
10.Add support for Units of Functionality to existing POSIX test suites	Modify current POSIX test suites so that they offer options to limit the test coverage to given units of functionality. These test suites would still require a runtime environment supporting POSIX .1 and .2
11.Modify TET for profiles 51 to 54	Develop a version of the TET framework that runs in restricted environments supporting only the subsets of POSIX .1 and POSIX .2 specified for each of profiles 51 to 54.
12.Modify VSXgen for profiles 51 to 54	Develop a version of the VSXgen test support environment that runs in restricted environments supporting only the subsets of POSIX .1 and POSIX .2 specified for each of profiles 51 to 54.
13.Review POSIX .1 assertions	Categorise existing POSIX .1 test assertions and create a set of test assertions valid for each of profiles 51 to 54, by checking all existing assertions for validity in each environment. Specify new test strategies for any assertions which are still testable in a given profile, but for which the existing test strategy is not supported in that profile
14.Develop test suites	Develop modular test suites for each of profiles 51 to 54 using the modified TET and VSXgen, and with test strategies that are fully supported by the profile.

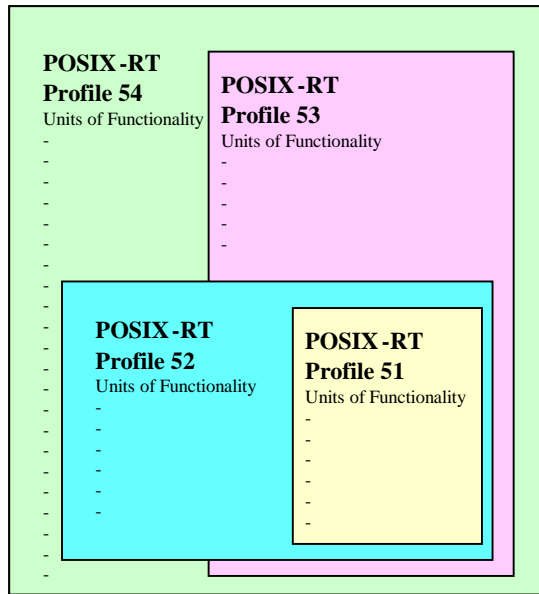
2.3 Technical Overview for Proposed Embedded Realtime System Testing

Embedded Realtime Systems run in a variety of hardware and operating environments. POSIX 1003.13 defines four different profiles for realtime applications. Some of these need not support a file-system or multiple processes. POSIX .13 also defines Units of Functionality which are subsets of the POSIX .1 specification that can be supported in isolation, without a system having to support the whole of the .1 standard.

A testing strategy for systems conforming to the .13 Profiles or Units of Functionality subsets needs to reflect real products. The TET/VSXgen framework cannot be used in its current form in the more restrictive profiles, since it makes certain assumptions about its operating environment that are not valid in these profiles. The Open Group's current POSIX test suites that use the TET/VSXgen framework may also not function in a profile in which the tests should apply, since a testing strategy may have been employed that lies outside the functionality of the given profile. The modifications needed to TET, VSXgen and the VSXgen-based POSIX test subsets in order to enable them to be used to test a given .13 profile are covered in this technical section.

2.3.1 Overview of POSIX 1003.13 Profiles

The POSIX 1003.13 Profiles 51 to 53 provide four levels of functionality to which realtime environments can conform. The profiles are based on a study of existing commercial practice, though most vendors have products that fall in a continuum covering the range of functionality that the profiles describe in snapshots.



**Portable Operating System Interface (POSIX)
IEEE Industry Standard**

All Profiles include some or all of POSIX .1, .1b and .1c, and some parts of POSIX .2 and .2a for the development platform (which is likely to be on a host system for Profiles 51-53). The Realtime profiles can be summarized as follows:

Profile 54:

All of POSIX.1, .1b, .1c .2, and .2a. Multi-process, Threads, File System.

Profile 53:

Multi-process, Threads, No File System.

Profile 52:

Single Process, Threads, File System.

Profile 51:

Single Process, Threads, No File System.

2.3.2 Overview of POSIX 1003.13 Units of Functionality

The POSIX 1003.13 Profiles make reference to other POSIX standards, but only a subset of the functionality specified by that standard. Where the referenced standard does not already have an option for specifying just the required functionality, only those Units of Functionality referenced by the profile may be used by a conforming application. The grouping of existing interfaces into Units of Functionality allows the further modularization of testing for compliance.

2.4 TET Modifications

This section describes how TET is to be modified to support the execution of test cases on systems that conform to the real-time profiles 51, 52 and 53 that are described in POSIX.13. Since these profiles do not support all the POSIX.1 functionality, it is desirable only to run a minimum amount of test harness code on the system under test in order to minimize the amount of code that must be changed.

2.4.1 Host and target systems

Execution of test cases that run on the system under test is controlled from another system; thus most of the TET code runs unchanged on that system. Only the code that runs on the system under test needs to be modified; the rest of the code (including tcc, the Test Case Controller) remains unchanged.

In the description that follows, the system under test is described as the target system and the system from which the test run is controlled is described as the host system. tcc runs on the host system.

2.4.2 TET version

The version of the harness to be used is Distributed TETware, since the client-server architecture employed in this version of TETware lends itself to the host system / target system approach.

It should be noted that the reason for using Distributed TETware is to minimize the amount of new code that must be written. It is not intended to support the running of distributed test cases on real-time systems.

2.4.3 Host system requirements

The system requirements for the host system are as follows:

- It must be capable of running Distributed TETware.
- If test cases are to be built on the hosts system, the host system should be capable of supporting the target system's development environment.

Ideally the host system will run the UNIX operating system.

If the host system is running Windows NT, it will not be possible to use TETware's test session interruption and test case timeout facilities to regain control of a hung target system.

2.4.4 Test case launcher

A new program provides the interface between the host system and the target system. Within the TETware architectural framework this program fulfills the role of an exec tool. This program is executed by tcc on the host system each time tcc wants to process a test case in execute mode.

This program includes the following functions:

- Download a test case image to the target system and instruct the system to execute it.
- Communicate with the running test case on the target system. These communications include:
 - Send information to the test case that is normally passed on the command line or in the environment.
 - Send information to the test case that is normally made available in files.
 - Receive test purpose results and other journal entries.
 - Receive notification of test case end.
- Forward test purpose results and other journal entries to tetxresd (the execution results daemon).
- Perform other administration functions on the target system.

This program runs on the host system and communicates with the TCM/API on the target system.

2.4.5 Target system software

This software is derived from the thread-safe version of the C TCM and API library, when built to support POSIX threads. To the extent possible, support is provided for the execution of non-distributed API-conforming test cases. Details of the API differences are presented in a later section.

There are two versions of this software. The difference between them relates to whether or not multiple processes are supported on the target system. Neither version requires file system support on the target system.

2.4.6 Communication between host and target systems

The host and target systems are connected together by a communication channel. Typically this will take the form of a serial link, although the use of a faster communication link (such as a network connection) is not precluded.

A simple message-passing protocol is implemented over the communication channel. It is not anticipated that this protocol will need to perform error correction.

On the target system, access to the communication channel is controlled by a mutex, so as to avoid contention in the event that more than one thread needs to use the link at once.

2.4.7 API versions

There are two versions of the API. One version is suitable for use on systems conforming to the profiles that do not support multiple processes (profiles 51 and 52), whereas the other version is suitable for use on systems conforming to the profile that supports multiple processes (profile 53).

All the non-distributed API functions and global variables are supported in each version except as noted below:

Profiles 51 and 52

tet_fork()

- The (*childproc)() function is called in a new thread, not in a new process. This has the following implications:
 - Static and global variables are shared between the (*childproc)() function and the rest of the process. They are not copies as would normally be the case after a real fork() operation.
 - The (*childproc)() function must not call exit() (either directly or by means of a call to tet_exit()). Instead it must return, or exit via a call to pthread_exit().
- If the (*childproc)() function returns, the API calls pthread_exit() with zero exit status.
- When tet_fork() interprets the validresults argument, it operates on the child thread's exit status. (Note: the pthread_exit() call takes a (void *) argument. In this description, the ``thread's exit status'' is the value pointed to by the argument to pthread_exit()).

tet_child

- This variable is not implemented.

pthread_t tet_child_tid

- This is a new variable provided by the API. It is a macro whose value is the contents of the address pointed to by the return value of a function. The address and contents are valid for the duration of a (*parentproc)() function. The value referenced by tet_child_tid is the thread ID of the child thread that is created by a call to tet_fork().

tet_exec(), tet_spawn(), tet_wait(), tet_kill()

- These functions are not implemented.

support for executed processes

- The child process controller (tcmchild.o, etc.) is not implemented.

Profiles 51, 52 and 53

tet_printf() and tet_vprintf()

- These functions make use of the null device. If there is no null device on a system and one of these functions is called to print a line which, after formatting, overflows the internal buffer (currently 16 Kb), the result is undefined. (If a null device is supported, its name must be specified using a compiler definition when TETware is built.)

tet_pname

- This variable is set to the name of the test case as it is known on the host system.

2.4.8 Names of the TCM and API components

All profiles

thrtcm.o	Test Case Manager
libthrapl.a	API library

Only on profile 53

thrtcmchild.o Child process controller

Note: these names include the suffixes that are used on UNIX systems. If the target system's development environment uses a different naming convention, suffixes appropriate to that environment are used instead.

2.4.9 User-defined functions

It is necessary for the exec tool and the TCM to perform several operations that are implementation-defined. Functions to perform these operations must be provided by the user for each combination of host and target system.

The functions to be provided include the following:

host system (exec tool)

- Open the control channel to the target system.
- Download a program image to the target system.
- Execute the down-loaded program.
- Close the control channel.
- Open the communication channel to the target system.
- Write a message to the communication channel.
- Read a message from the communication channel.
- Close the communication channel.
- Terminate execution of the program running on the target system (see ``Test case termination" below.)
- Restart (or reboot) the target system.

target system (TCM)

- Open the communication channel from the host system.
- Write a message to the communication channel.
- Read a message from the communication channel.
- Close the communication channel.
- Reset implementation-defined elements of the process environment to the default state.
- Terminate the program (see ``Test case termination" below.)

It may be that the need for other functions is identified during development.

2.4.10 TCC - TCM interface

In the ``standard" version of TETware, tcc passes the following information to a test case:

- When tcc executes the test case, the command-line contains the test case name and the list of ICs to be executed.
- Certain communication variables are placed in the environment:

TET_ACTIVITY	tcc's activity counter
TET_CODE	name of the file containing result codes
TET_CONFIG	name of the file containing configuration variables
TET_EXECUTE	the name of the alternate execution directory (if specified)
TET_ROOT	the name of the TET root directory
TET_RUN	the name of the runtime directory (if specified)
TET_SUITE_ROOT	the name of the alternate directory below which test suite root directories may be found (if specified)
TET_TIARGS	miscellaneous control information
TET_TSARGS	information about Distributed TETware servers

- Configuration variable assignments are made available in a file.
- The list of result codes are made available in a file.

The test case passes the following information back to tcc:

- Journal information and test purpose results are written to a file (in TETware-Lite) or sent to tetxresd (in Distributed TETware).
- The test case exit status.

In the ``restricted profile" version of TETware, this interface is modified as follows:

- tcc invokes the exec tool with test case name and IC list as arguments.
- The exec tool down-loads the executable test case image to the target system and instructs the target system to run the image.
- The exec tool sends the following information to the TCM:
 - test case name
 - IC list
 - TET_ACTIVITY value
 - TET_TIARGS value

(Note that the values of communication variables are not put in the environment on the target system.)

- The exec tool sends the set of configuration variable assignments and result code definitions to the TCM, which stores them in memory on the target system.
- In profile 53 the TCM sends its process ID to the exec tool.
- The TCM sends the following message types to the exec tool for forwarding to tetxresd on the host system:
 - Test Case Manager Start
 - Invocable Component Start
 - Invocable Component End
 - Test Purpose Start
 - Test Purpose Result
 - Test Case Manager Message
 - Test Case Information
- The TCM sends an exit status message to the exec tool immediately before terminating.

2.4.11 Making journal entries

The API functions in this category are modified internally to use the communication channel, rather than writing to files or sending messages to tetxresd. However, the program interface remains the same as in the ``standard" C API.

2.4.12 Accessing configuration variables

The tet_getvar() API function is modified internally to access configuration variable assignments that are store in memory by the TCM, rather than reading the variables from a file. However, the program interface remains the same as in the ``standard" C API.

2.4.13 Detecting test case malfunction

After the TCM has received the information from the exec tool (as described in a previous section), but before it calls the user-defined test case startup function, it starts a new thread. This thread is known as the "heartbeat thread". The purpose of this thread is to send messages at (say) one second intervals to the exec tool, so that the exec tool knows that the test case is still running normally.

If the exec tool doesn't receive a message after a suitable period of time has expired, it interprets this to mean that the test case has hung or terminated abnormally. It writes a suitable message to the journal, calls the user-defined test case termination function and exits with a non-zero exit status.

2.4.14 Test case termination

In profiles 51 and 52 there is no exit() function. Thus it is necessary to cater for situations where normal program termination can be performed on the target system, or where termination must be accomplished by performing some action on the host system.

When a test case has finished execution, it sends an exit status message to the host system. On systems conforming to profiles 51 and 52, the TCM calls the user-defined termination function. On a system conforming to profile 53 the TCM calls exit().

When the exec tool on the host system receives the test case termination message:

- When testing systems conforming to profiles 51 and 52 the exec tool calls the user-defined test case terminate function with an argument that indicates that the test case has terminated (or wishes to terminate) normally.
- When testing systems conforming to profile 53 the TCM does not call the user-defined test case terminate function.

If the exec tool on the host system needs to force a test case running on the target system to terminate, it calls the user-defined test case termination function with an argument that indicates that the test case must be terminated forcibly. On profile 53 the test case's process ID is also passed to the test case termination function.

2.5 VSXgen Modifications

In order to provide an infrastructure for the existing POSIX .1 test suites, and those specified earlier in this document, the VSXgen package needs to be modified.

- The distinction between host (compiler) and target (runtime) environments needs to be made in the VSXgen configuration. Many services will not be needed in the target environments when cross compiling, so these will not be built into the target libraries.
- Support functions implemented using strategies that are not available in a given profile will have to be modified or removed.

Detailed descriptions of the changes needed to VSXgen in order to support cross-compilation, and to restrict strategies to those available in a given profile, are given in the next sections.

2.5.1 Changes Needed for Cross-compilation of VSXgen

Within VSXgen there are a number of utilities that are built from C source code. Currently these are all compiled using the same compiler commands as used when building the executable tests. In a cross-compilation environment these utilities need to be built using the native compiler on the development system, and the executable tests need to be built using the cross-compiler. There are a few other places where compilation commands are used other than for building libraries, utilities and test executables (e.g. where the -E option is used). In some of these cases it will be appropriate to use the cross-compiler and in others the native compiler.

In order to accommodate the use of two different sets of compiler commands and related options, substantial changes are needed to the configuration and installation scripts and procedures in VSXgen. Changes are also needed to the header test driver to allow the individual test programs it creates to be executed on the target system.

2.5.1.1 Configure and Install Stages

The config.sh script will ask the user whether the test executables are to be cross-compiled. If the answer is yes, it will ask for information related to the cross-compiler. This will be used to set the new parameter values XCC, XCOPTS, XLDFLAGS and XSYSLIBS in SRC/vsxparams. When the questions related to the existing parameters CC, COPTS, etc. are subsequently asked, the script will make it clear that the questions relate to the native compiler.

The 'test mode' selected by the user should only apply to code which is used in test executables or which affects the execution of tests. This is because it cannot be assumed that the development system has the same level of compliance with the POSIX/UNIX standards as the target system. Only POSIX.1-1990 interfaces should be used on the development system (plus some of the usual user-supplied interfaces).

The header file checking performed during the configuration stage will be done using the cross-compiler only. Since the native libraries and utilities only use POSIX.1-1990 interfaces it is unlikely that anything would be found to be missing from the headers. In any case compilation failures involving the native compiler can only occur when libraries and utilities are built, which is during the install stage, so not much is lost by not detecting these problems during the configuration stage.

The header files placed in SRC/SYSINC will be copies of the headers for use with the cross-compiler. This is necessary in order for the dependencies in testset Makefiles to work correctly.

Some libraries will need to be built in two versions - one native and one cross-compiled. The one with the 'normal' name will be the cross-compiled one, so that testset Makefiles do not need to be changed. The native versions will be compiled in a separate directory with a "nat_" prefix, in a similar manner to the way the threads versions of the libraries are compiled, and the installed file will also have the "nat_" prefix. The Makefiles used for these compilations will not use the SYSINC header file copies.

All of VSXgen's utilities except purefile will be built using the native compiler. Again, these compilations will be done in a separate "nat_" directory. The Makefile will not use the SYSINC header file copies, and will use the "nat_" libraries. The vbuild utility and the header test driver will be linked with the native versions of the TET TCM and API library.

Because the libraries and utilities in the 'normal' locations are the cross-compiled ones, and the Makefiles in these directories use the normal variable names \$(CC), \$(COPTS), etc., the install script will have to set these variables to the corresponding cross-compiler parameter values when it creates each Makefile from the Makefile.org template. It needs to know that it should not do this when compiling native libraries and utilities, and it will use a test for the "nat_" prefix on the directory name to determine that the normal values should be used. This mechanism will also apply to libraries and utilities from VSXgen packages, thus allowing packages which have been updated to support cross-compilation to control how the compilations are done by the install script.

Two different versions of userintf.c will be needed, since the code to implement the user-supplied functions may need to be different on the native system and the target system. However, only very few functions will be needed in the native version. It is probably fairly safe to assume that only the functions related to "appropriate privilege" are needed, so (initially at least) there will be no facility for packages to specify additional functions to be placed in the native version of userintf.c.

Initially at least, all facilities related to the wide character locales will not be available in cross-compilation configurations.

The user-configurable tasks normally carried out in the top-level Makefile may perhaps not be workable in some cross-compilation configurations. In this case the tasks can be carried out manually on the target system. (The user guide already suggests this as an alternative means of carrying out the tasks.)

When the install script creates tetexec.cfg, it will set the compiler-related parameters (VSX_CC, VSX_CFLAGS and VSX_LIBS) appropriately for cross-compilation.

2.5.1.2 Build Stage

The `vmake.sh` script will arrange for the cross-compiler to be used by "make". It will do this by setting the usual variable names (`CC`, `COPTS`, etc.) on the "make" command line to the corresponding cross-compiler parameter values instead of the normal values, so that the testset Makefiles do not need to be changed. It will also set the TET-related variables (`APILIB`, `TCM`, etc.) to the locations of the cross-compiled versions of the TET TCM and API library.

When `vmake.sh` uses a compiler command to detect whether a macro version of any of the tested interfaces exists, it will use the cross-compiler. If no macro exists, `vmake.sh` creates a "dummy" test executable which produces `NOTINUSE` results. Currently this dummy test executable is a shell script. It will either need to be a binary (built with the cross-compiler) or perhaps a method could be employed to indicate to the `TET_EXEC_TOOL` that it should execute the shell-script dummy executable on the development system instead of on the target system. The latter would have the advantage of not wastefully compiling and executing a lot of tests which will end up just producing `NOTINUSE` results.

2.5.1.3 Execution stage

All testset executions will be done via a `TET_EXEC_TOOL`, which will be responsible for loading cross-compiled test executables onto the target system and waiting for their completion. Not all testsets are cross-compiled, and so the `TET_EXEC_TOOL` will decide, based on criteria such as the section name of the testset, whether the test executable should be executed on the development system (e.g. for header test sections) or on the target system (probably for all other sections). The section name alone would be enough to ensure that for header tests the driver is executed on the development system, but for other cases additional criteria may also need to be considered - see the discussion above about dummy macro testsets.

2.5.1.4 Header tests

When a header testset is executed, the test executable is installed as a link to the header test driver. This is a TET API-compliant testcase which extracts some test code for each test from an archive and then compiles and (optionally) executes the test code for each test in turn. Therefore the issues already discussed above in relation to the building and execution of interface tests also apply to the header tests, but must be addressed within the header test driver itself.

Since the parameters `VSX_CC`, `VSX_CFLAGS` and `VSX_LIBS` will have been set appropriately for cross-compilation, this means no change is needed to make the header driver invoke the cross-compiler.

Execution of the individual tests will need to be done in a similar manner to the way the `TET_EXEC_TOOL` executes the non-header tests. Perhaps the header test driver and the `TET_EXEC_TOOL` can both make use of a common underlying tool. Or perhaps the `TET_EXEC_TOOL` can be made versatile enough to perform both functions itself.

Currently the header tests report failures to `stdout` and this is captured by the header test driver. The test code uses a "PRINT" macro, defined in `HEADER.h`, to write to `stdout`. This macro will need to be changed so that the information is sent over the link to the header test driver (or the `TET_EXEC_TOOL`) instead.

Another complication is that currently the result of each test is communicated to the header test driver via the exit status of the process. When the tests are run on the target system the result will have to be sent over the link instead. Unfortunately the tests do not use a macro to report the result - they just call `exit()` directly or use a "return" statement in `main()`. This means that all header tests which need to be able to run in a cross-compilation environment will have to be modified so that they report the result code using a new macro (or function) defined in `HEADER.h`.

2.5.1.5 C compiler tests

It is assumed that the C-compiler test driver (`driver.C`) will not be needed in cross-compilation environments. The only tests that use this driver are the "XPG3 C Language" tests in `VSX4`.

2.5.2 Changes Needed to VSXgen to Support Testing of POSIX.13

VSXgen contains a number of libraries that are used by testset executables. When these libraries are compiled for use by tests of the PSE51, PSE52 or PSE53 realtime profile, the code that is visible to the compiler must not make use of any interfaces that are not required in the profile.

An initial analysis of the library code has been undertaken and the findings are presented in the next section. This analysis is based only on which interfaces are called. It does not address finer constraints such as use of kill() with a negative argument.

The changes discussed here are solely concerned with the interfaces used by VSXgen library routines. There are additional changes to VSXgen needed in order to support cross-compilation, presented above.

Some issues with POSIX.13 have arisen from the analysis, and these are discussed in a separate section following the results of the analysis.

2.5.2.1 vport Library

nonposix.c:

The popen(), pclose() and system() functions will not be defined when testing profiles PSE51 and PSE52. (They are probably of no use for PSE53 either). The strtol() function will not call setlocale() when testing profiles PSE51 and PSE52.

vargtypes.c:

The vgetcwd() function will not be defined when testing profiles PSE51 and PSE53.

2.5.2.2 vlib Library

arc_extr.c:

The functions in this file will not be defined when the file is cross-compiled. (They are not used by test executables.)

debug.c:

The debugging facility either cannot be provided when testing profiles PSE51 and PSE53, or it will have to send the debug output to the development system instead of writing it to a file. If the latter, then the calls to access(), chmod(), chown(), exit(), fcntl(), getgid(), getpid(), and getuid() in debug.c will need attention.

error.c, panic.c:

The vsxerror() and panic() functions report errors to stderr. They will have to send the error messages to the development system instead.

report.c:

The reporting functions call exit() if tet_vprintf() returns an error. For profiles PSE51 and PSE52 they will need to use an alternative means of terminating execution of the testset, perhaps by calling abort().

2.5.2.3 genlib Library

childsync.c, closesync.c, opensync.c, parsync.c, sendsync.c, waitsync.c:

The synchronisation functions defined in these files make use of a pipe. When testing profiles PSE51 and PSE52 they will have to be rewritten to use a different synchronisation method.

chktimes.c:

None of the functions in this file will be defined when testing profiles PSE51 and PSE53. When testing profile PSE52 the sleep() calls will be changed to use a different means of sleeping, perhaps alarm() and pause().

clean_dir.c:

The `clean_dir()` function will not be defined when testing profiles PSE51 and PSE53. When testing profile PSE52 it will not call `chmod()`.

crdir.c:

The `crdir()` function will not be defined when testing profiles PSE51 and PSE53. When testing profile PSE52 it will not call `chmod()` and `chown()`, and will ignore the mode, uid and gid arguments.

crfile.c:

The `crfile()` function will not be defined when testing profiles PSE51 and PSE53. When testing profile PSE52 it will not call `chmod()` and `chown()`, and will ignore the mode, uid and gid arguments.

eaccess.c:

The `eaccess()` function will not be defined when testing profiles PSE51 and PSE53. For profile PSE52 the function can just call `access()`. (The existing code cannot be used because it calls `geteuid()` and `getegid()`.)

fpcomp.c:

The code (included from `FPCOMP.h`) can report an error to `stderr`. It will have to send the error message to the development system instead. Care must be taken not to interfere with the use of `FPCOMP.h` in header tests.

killchild.c:

The `killchild()` function will not be defined when testing profiles PSE51 and PSE52.

vxmkdir.c:

The `vxmkdir()` function will not be defined when testing profiles PSE51 and PSE53.

2.5.2.4 tsetlib Library

do_access.c.c:

None of the functions in this file will be defined when testing profiles PSE51, 52 and 53.

do_feio.c:

None of the functions in this file will be defined when testing profiles PSE51, 52 and 53.

do_ferr.c:

The `do_ferr()` function is used to set up conditions for testing stdio errors. For some of the errors it creates a pipe, FIFO or plain file, and for others it creates a child process. The function will need to be modified so that, when testing profiles PSE52 and PSE53, only those parts that are appropriate for the selected profile are used. When testing PSE51 most of the errors are likely to be considered untestable. For those that are testable (probably just EBADF) a different strategy will have to be used. Also, when testing PSE52, the `geteuid()` and `getegid()` calls used as arguments to `crfile()` will need to be changed to dummy values.

mountfs.c:

None of the functions in this file will be defined when testing profiles PSE51 and PSE53. For profile PSE52 it is likely that many implementations will not provide the means to mount an additional file system, so the `mountfs()` function will need to allow for the `VSX_MOUNT_DEV` parameter to be set to "unsup".

nametoolng.c:

None of the functions in this file will be defined when testing profiles PSE51 and PSE53.

n_open.c:

The `n_open_fds()` function uses `fstat()` to determine which file descriptors are open. When testing profiles PSE51 and PSE53 it will need to use a different function for this purpose. One possibility might be `fchmod()`, unless a more suitable alternative can be found.

pathres.c:

None of the functions in this file will be defined when testing profiles PSE51 and PSE53. When testing profile PSE52, the `getuid()` and `getgid()` calls which are used to obtain argument values for `crfile()` will be replaced by dummy values.

pipeclib.c, pipeplib.c:

If any tests that use the "pipe reporting" functions provided by these files are considered testable for profiles PSE51 and PSE52, then the functions will have to use a different communication mechanism instead of a pipe.

pure_run.c:

The functions in this file will not be defined when testing profiles PSE51, 52 and 53.

setupnospc.c:

None of the functions in this file will be defined when testing profiles PSE51 and PSE53. If any `ENOSPC` assertions are considered testable for profile PSE52, then either this function will require major reworking, or the tests that use it will need to be rewritten to use a different setup method.

setupprofs.c:

None of the functions in this file will be defined when testing profiles PSE51 and PSE53. If any `EROFS` assertions are considered testable for profile PSE52, then either this function will require major reworking, or the tests that use it will need to be rewritten to use a different setup method.

statdevind.c, statmode.c, statugid.c:

The functions defined in these files are just wrappers around `stat()` or `fstat()`. They will not be defined when testing profiles PSE51 and PSE53.

wait_child.c:

The `wait_child()` function will not be defined when testing profiles PSE51 and PSE52.

termios.c:

None of the functions in this file will be defined when testing profiles PSE51, 52 and 53.

tztest.c:

There is a `getenv()` call used in this file. This will be affected by the issue regarding `getenv()` and "environ" which is discussed under "Issues with POSIX .13" below.

2.5.2.5 tet_startup

startup.c:

Normally the `startup()` function creates a "working directory" and changes directory to it. The `cleanup()` function changes directory out of the working directory and then removes it. When testing profiles PSE51 and PSE53 none of this will be done.

The `cleanup()` function will not make calls to `killchild()` when testing profiles PSE51 and PSE52.

2.5.3 Issues with POSIX .13

Initial issues identified with POSIX .13 are listed below. Further issues are expected to be raised during the development process.

2.5.3.1 fstat()

It seems odd that `fstat()` is not mandatory in all profiles, since it can be used to obtain information about shared memory objects, and the `_POSIX_SHARED_MEMORY_OBJECTS` option is mandatory in all profiles. Any future changes to the requirements regarding `fstat()` will affect the changes listed for `tsetlib/stat*.c` and `tsetlib/n_open.c`.

2.5.3.2 getenv() and environ

In POSIX.13 the function `getenv()` is not mandatory in all profiles. However, no mention is made of the global variable "environ", so it is not clear whether this is available in profiles PSE51, 52 and 53. This will affect tests of `execl()` and `execve()`, and also `tzset()` and all the functions from the C Standard that are required to have the effect of a call to `tzset()`.

There seem to be four possible outcomes:

1. The "environ" variable is only available in profiles PSE53 and PSE54 (where `getenv()`, `execl()` and `execve()` are mandated).

In this case it will not be possible, in profiles PSE51 and PSE52, to perform any tests which involve changing the value of TZ. The functions defined in `tsetlib/tztest.c` will not be provided for these profiles.

2. The "environ" variable is only available in profile PSE54.

This seems unlikely, but if it is the case then, as well as tests involving TZ being affected as in 1 above, some assertions for `execl()` and `execve()` will not be testable for profile PSE53.

3. The "environ" variable is available in all profiles.

The absence of `getenv()` in some profiles, in this case, can be addressed by providing a `getenv()` function in `vport/nonposix.c` when testing profiles PSE51 and PSE52. A declaration will also be needed in `sysdep.h` in case it is not declared in `<stdlib.h>`, but preceded by a `#undef getenv` in case it is defined as a macro in `<stdlib.h>`.

4. The "environ" variable is available in all profiles, and a future revision of POSIX.13 makes `getenv()` mandatory in all profiles.

In this case no changes related to `getenv()` and "environ" are needed.

2.5.3.3 Support for Test Locales

Although the `setlocale()` function is mandated for profile PSE53, since support for POSIX.2 is not required, it is not clear how (or indeed whether) the VSXgen test locales can be installed on the target system. It may be necessary to modify the tests which use these locales so that they can give an Unsupported result (or omit parts of the test) if the VSXgen test locales are not available.

2.6 Test Suite Modifications

The changes to the test framework to enable it to function in all POSIX .13 profiles provide a platform on which the existing test suites for POSIX developed by The Open Group can potentially be run. In order to produce an appropriate set of tests that use support test strategies for each profile, the process outlined in the following sections will be followed.

2.6.1 Categorize All POSIX Assertions

The existing POSIX test suites, and those specified in the earlier sections of this document, are generated from sets of assertions. Each existing assertion must be analyzed to determine which profiles it applies to, if any. This will involve checking if the API referred to by an assertion is required, and if so, whether the particular semantics of the API that the assertion addresses still apply in the profile.

2.6.2 Determine Testability and Strategies of Applicable Assertions

For all assertions that apply to each profile, it must be determined whether the current strategy can still be used in that profile. There may be cases where only part of a strategy applies to a given profile, e.g. file related functions that are tested using regular files, FIFOs, pipes and terminal devices will not be able to use the whole range of file types on each profile. If the current strategy cannot be used in a profile, it must be determined whether a new strategy can be developed using functionality that is required by that profile, and thus guaranteed to be available on all conforming platforms.

Assertions for which there is no alternative strategy in a given profile will be untestable in that profile. It may be that an assertion that is testable in a larger profile, or in a full UNIX system, is not testable in a smaller profile due to constraints on the test strategies that can be adopted in that profile.

2.6.3 Implement New Strategies for Testable Assertions Where Required

For all assertions on a given profile that are testable, but using newly developed strategies, new test cases will need to be developed. For all tests for which existing strategies are still broadly applicable, checks and changes will need to be made in a similar manner to those described for VSXgen, above, to ensure that the interfaces used are supported on the profile.

2.7 Current Commercial Practice

The Open Group is in consultation with leading commercial suppliers of realtime operating systems and embedded products. The information gained in discussions with these suppliers has formed the basis for the proposed modifications and costs for the test suites and framework. A brief summary of the issues discussed with the suppliers is presented below.

2.7.1 Product Diversity

The range of products that fall within the categories addressed by POSIX .13 is huge, and there is great diversity in hardware and software. A wide variety of hardware platforms are used, ranging from PCs to onboard ROMs. Quoted memory sizes ranged from 512Kb to 64Mb, and operating system sizes ranged from 50Kb to around 1Mb. Platforms typically had more memory available for applications than was used by the operating system, which should allow the test suites, which will run as applications on the platforms, to have sufficient memory. Target hardware, at least during development, has at least a serial port for communication to and from a host environment, and in many cases networking support is available.

2.7.2 Development environments

The development environments used for realtime platforms range from self-hosting compilations in POSIX.2-compliant environments to cross compilation from server platforms. In the cross compilation environments the ease and amount of automation involved in downloading the operating environment to the target varied considerably. Since the test suites will need to be downloaded onto the target in an automated way, there must be user-supplied code to perform the test downloads. In environments where the application and kernel must be linked together on the host and downloaded to the target over a serial line, possible once per test purpose, the tests may take a number of days to execute.

2.7.3 Process Execution

The mechanisms used to start and stop processes, and the result of process termination, are varied and no particular behaviour can be guaranteed. User-supplied code must be used to start tests, return results from them, and re-initialise the environment between tests.

END OF DOCUMENT